NASA/CR-2001-211224
ICASE Interim Report No. 38

ICASE Workshop on Programming Computational Grids

Thomas M. Eidson and Merrell L. Patrick
ICASE, Hampton, Virginia

ICASE
NASA Langley Research Center
Hampton, Virginia

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

September 2001

Report Documentation Page

Report Date 00SEP2001	**Report Type** N/A	**Dates Covered (from... to)** -

Title and Subtitle ICASE Workshop on Programming Computational Grids	**Contract Number**
	Grant Number
	Program Element Number
Author(s) Thomas M. Eidson and Merrell L. Patrick	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23681-2199	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)

Distribution/Availability Statement
Approved for public release, distribution unlimited

Supplementary Notes
ICASE Interim Report No. 38

Abstract
A workshop on Programming Computational Grids for distributed applications was held on April 12{13, 2001 at ICASE, NASA Langley Research Center. The stated objective of the work- shop was to de ne, discuss, and clarify issues critical to the advancement of Problem Solving Environ- ments/Computational Frameworks for solving large multi-scale, multi-component scienti c applications us- ing distributed, heterogeneous computing systems. This report documents a set of recommendations for NASA that suggest an approach for developing an application development environment that will meet future application needs.

Subject Terms

Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** SAR

Number of Pages
17

ICASE WORKSHOP ON PROGRAMMING COMPUTATIONAL GRIDS[*]

THOMAS M. EIDSON[†] AND MERRELL L. PATRICK[‡]

Abstract. A workshop on Programming Computational Grids for distributed applications was held on April 12–13, 2001 at ICASE, NASA Langley Research Center. The stated objective of the workshop was to define, discuss, and clarify issues critical to the advancement of Problem Solving Environments/Computational Frameworks for solving large multi-scale, multi-component scientific applications using distributed, heterogeneous computing systems. This report documents a set of recommendations for NASA that suggest an approach for developing an application development environment that will meet future application needs.

Key words. software components, computational frameworks, scientific applications, computational grids, distributed computing

Subject classification. Computer Science

1. Introduction. A workshop on Programming Computational Grids for distributed applications was held on April 12–13, 2001 at ICASE, NASA Langley Research Center. Twelve researchers, software developers, and users of Problem Solving Environments/Computational Frameworks from government and university laboratories participated. The stated objective of the workshop was to define, discuss, and clarify issues critical to the advancement of Problem Solving Environments/Computational Frameworks for solving large multi-scale, multi-component scientific applications using distributed, heterogeneous computing systems.

As part of the preparation for the workshop, a small panel of experienced application developers was assembled and recommendations for programming needs were discussed. The results of these discussions were presented to the workshop attendees. During the discussion of requirements and other issues with both the application developers and the system software developers, it was clear that neither group fully understood the ideas and problems of the other. It was also clear that neither group is given the time and support to investigate the requirements of modern applications programming and to translate those requirements into design requirements for Problem Solving Requirements. Notwithstanding, these discussions led to a set of recommendations for NASA managers and application developers who need to use computational frameworks to solve their multi-disciplinary scientific applications. While targeted for NASA, the recommendations should be of interest to the entire scientific programming community.

Following the Introduction, the report opens with a set of definitions. The primary focus of the report is the recommendations to NASA which are presented in Section 4 of the report. In Sections 5 and 6, several technical issues relating to scientific programming requirements are presented, which are intended to augment the recommendations. While there was a general consensus that software component technology offers a significant potential, there is disagreement on specific requirements of scientific applications. The result is that the development of prototype frameworks as well as general research into scientific software components lacks focus. Sections 5 and 6 of the report elaborate on some of the issues that need to be

[*]This workshop was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681-2199.

[†]ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23681, teidson@icase.edu

[‡]ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23681, mpatrick@icase.edu

addressed before the research community can move to a more focused approach.

The technology direction suggested in this report is already being taken by some researchers at NASA, as well as others in the scientific community. The report recommends more focus and co-ordination. A set of these related research projects are summarized in the Appendix.

2. Definitions. The programming technology discussed in this report is in an evolving research stage. As such, there are few terms with definitive definitions. The following definitions were included to provide clarity to the discussions that follow.

A *Problem-Solving Environment* (PSE) is an integrated collection of software tools that facilitates problem-solving in some domain. This includes defining, building, executing, and managing the application. Additionally, this can include viewing and analyzing results related to the problem being solved.

A *computational framework* is an integrated collection of software tools that facilitates the development and execution of an application. A framework is the core feature of some PSEs.

A *programming model* is a set of abstractions and a set of rules that specify the combination of those abstractions in a form that can be translated to create execution instructions for an application.

A computational *Grid* is a collection of heterogeneous computational hardware resources that are distributed (often over a wide area) and the software to use those resources. An important feature that converts a set of computers and software connected by an internet into a Grid is a set of support services (resource management, remote process management, communication libraries, security, monitoring support, etc.) and an organizational structure that provides usage guidelines or rules.

Grid programming is just a subset of *distributed programming*. Distributed programming initially was focused on developing applications distributed on a local area network (LAN) where administration and security problems were minor. Also, the heterogeneous nature of the computers on the LAN typically covered a narrow range. Grid programming just extends distributed programming into more complicated, wide-area, heterogeneous environments.

An *element application* is a code in stand-alone executable or library form, that is focused on a relatively narrow aspect of some physics, mathematics, graphics, or other science. Sometimes an element application corresponds to some scientific discipline; thus, element applications are sometimes called discipline applications.

A *composite application* is defined as an application that is developed from the integration of smaller element applications. Examples of composite applications are (i) codes built from numerical libraries and (ii) a design code that integrates several discipline codes along with an optimization code.

Metadata is information about some programming entity that supports its use in some more comprehensive program (or *meta-program*) such as a composite application. Metadata includes interface specifications that describe how to access the programming entity and behavioral specifications that describe conceptual and practical details of correctly integrating the entity into the meta-program. For example, the information expressed in a Fortran subroutine could define some numerical algorithm. Interface metadata would describe the arguments needed to call that subroutine, typically in some general language. Behavioral metadata might describe the parallelization strategy as it relates to target machines. Behavioral metadata could even be used to describe physical and numerical assumptions embedded in the numerical algorithm.

A *software component* is a basic unit of software packaged for use in efficiently building some larger composite application. The software package includes metadata that minimally defines any interfaces to that software so that some computational framework can more easily provide the necessary integration. Software component technology is intended

- to support software reuse and sharing,
- to simplify use of multiple languages,
- to support the efficient building of large applications, and
- to assist building distributed applications.

3. Related Forums. Two key scientific community-based groups that are playing a leadership role in the development of software component and computational grid technologies are described. These organizations play a role in the implementation of the workshop's recommendations.

The Global Grid Forum (Global GF) [3] is a community-initiated forum of individual researchers and practitioners working on distributed computing or Grid technologies. Global GF is the result of a merger of the Grid Forum, the eGrid European Grid Forum, and the Grid community in Asia-Pacific. Global GF focuses on the promotion and development of Grid technologies and applications via the development and documentation of "best practices," implementation guidelines, and standards with an emphasis on rough consensus and running code. Efforts are also aimed at the development of a broadly based Integrated Grid Architecture that can serve to guide the research, development, and deployment activities of the emerging Grid communities. Such an architecture will advance the impact of the Grid through the broad deployment and adoption of fundamental basic services and by sharing code among different applications with common requirements.

The Common Component Architecture Forum (CCA Forum) [4] is a group of government lab (mainly DOE) and university researchers whose objective is to define a minimal set of standard features that a high-performance component has to provide, or can expect, in order to be able to use components developed within different PSEs. Such standards will promote interoperability between components developed by different teams across different institutions.

4. Recommendations for Future Direction. Because of the unprecedented increase in both single platform and distributed computing capabilities, scientific computer applications are evolving to tackle much larger, multi-scale problems where the simulation or modeling of a range of different physics needs to be solved, often as a coupled system. Emerging Grid technologies for wide-area distributed computing provide the foundation for these large-scale applications to use internets in building a computational infrastructure for their solutions.

Historically, scientific application developers have exhibited a great deal of independence in the programming styles they used while focusing on the need for creativity and persistent experimentation of their disciplinary codes. Given the multi-disciplinary and distributed nature of the applications, the current need is to focus on technology transfer to reap more of the benefits of years of software research and development of disciplinary codes, software libraries, and tools. The result is that programming efficiency, code maintenance, code clarity, and code sharing with performance guarantees have become more important. This means that modern programming methodology and practices need to be focused on good organization, flexibility, adaptability, and re-usability. Such practices should support portability and interoperability of codes. The recommendations and associated discussions below target these needs.

- Recommendation 1

  A scientific programming model for developing and executing composite applications should be based on software component technology.

  A software component is just a way of packaging code in a modular manner with clearly defined interfaces. Metadata is included as part of the component to provide details that enhance the integration of that code into an application. The primary target of a software component is a framework that understands the packaging protocol and the metadata to provide a component integration environment. If the packaging and metadata specifications are appropriately designed, a code packaged in component form will contain information about its use that has value outside a targeted, component-based framework. This means that the component methodology can co-exist with other programming systems. An ideal goal would be for the organizational and packaging characteristics to be pervasive in the programming community while supporting alternative programming approaches, not preventing them.

– Advantages of Software Component Technology
  1. A key design feature of software component technology is the rapid integration of an element code (or component) into a composite application.
  2. Code maintenance and validation will be easier because of the modular design and formal packaging requirements of the component approach.
  3. Codes maintained in component form will tend to be reused because of the above advantages. This includes not only reuse by the code developer, but efficient sharing of software with others.
  4. Distributed programming can be easier and more efficient when based on component technology. Distributed software development includes both conceptual and practical issues. The conceptual issues—distributed layout, synchronization, control flow, data flow—are often easy for programmers to define. However, the practical issue of managing the many details can overwhelm many programmers. These details include:
     ∗ managing distributed files,
     ∗ managing code at different sites with different computers and different architectures,
     ∗ determining which code will run where, and
     ∗ managing complex control flows.

     Software component technology can provide a systematic format that assists programmers in solving each issue in a step-by-step procedure.
  5. Metadata associated with a component (integrated specifications and documentation) can result in increased confidence in the modified application created by changing or adding a component.
  6. Code portability can be improved; e.g., a distributed computing solution can be used to run a code on its "natural" architecture, rather than converting it to run on the user's desktop.
  7. Computational frameworks based on plug-and-play components support rapid prototype and production code development.
  8. The need to understand and to use multiple languages is reduced. Frequently, users want to use a code that is written in an unfamiliar language. Use of generic interfaces reduces the need to learn the interface details of such codes.

– Costs of Software Component Technology
  1. Programmers will need to learn new tools and to develop new programming practices. The long-term benefits resulting from the use of the resulting tools should offset the overhead associated with learning new tools and practices. An important issue is to involve application programmers and users in the tool design so that the most appropriate tools are developed.

2. Some programmers will need to learn new programming styles. However, the component programming style focuses on good code and data organization to enhance understanding an application and to minimize data motion. Such coding practices are already in use by many application developers.
3. Program management will need to be convinced that the resulting benefits will justify the transition costs. In many cases, current programming practices simply do not support management objectives. Complex composite applications are needed to solve many modern engineering problems. These applications need significant increases in functionality with decreased software production costs.
4. Legacy codes will require some redesign. However, the component approach does allow for an incremental redesign strategy. For example, the most valuable kernels can be converted to components first. As more elements become available in component form, it will be cheaper to rebuild a complex legacy code from components as compared to maintaining it.
5. Large application systems with complicated couplings between code elements will be the hardest to redesign. But even here, the incremental approach will eventually become effective.
6. The component approach does result in some performance overhead to support its flexibility. Appropriate design of frameworks will minimize this. In the long run, features such as rapid prototyping can actually result in better delivered performance as more designs can be tested when programming time is limited. Most importantly, component designs free the author of a particular component to focus most of his efforts on creating optimized versions for different situations, rather than spending time maintaining infrastructure codes with no impact on performance.
7. The component approach may prevent compilers and runtime systems from performing cross-module optimization. On the other hand, compilers could evolve and use the meta-data associated with components to better handle these optimizations.

- Recommendation 2

  NASA should form a task force of software and application developers along with potential users to provide computational framework requirements and work with the CCA Forum developers. It is critically important that NASA provides its requirements to both the CCA and Global Grid Forums.

  The design and implementation of an effective component-based framework will need input from application developers and users. Application scientists should help to define framework specifications by identifying the requirements for components and services that will be needed to build their applications. Additionally, application developers need to suggest programming models that will be most understandable and that will be efficient to use. Any framework specifications should support one or more such programming models. Providing a close relationship between software developers and users will speed up the development of a quality product.

  The Common Component Architecture (CCA) Forum offers the best starting point for coordinating the development of scientific component technology. The CCA specification offers an attractive starting point because its design focuses on specifying the minimum essential elements of a component programming environment. This will allow the scientific community to develop the higher abstractions to best suit their needs. The CCA Forum could be viewed as focusing on a limited, but extremely important element of the complete Grid programming problem that is the focus of the Global Grid Forum. This element, developing the core of a programming model and framework infrastructure, provides a foundation for efficient use of the Grid and, thus, will require interaction between the CCA Forum and all the working groups in the Global Grid Forum. Specifically, the CCA Forum could focus on demonstrating the viability of prototype and/or standard implementations of the CCA specification. It is important to show that a CCA-compliant framework can be delivered as an open source platform that will work with all Grid standards. It should be easily downloaded and installed.

- Recommendation 3

  NASA should join with other mission-oriented government agencies, maybe through the National Coordinating Office for Information Technology Research and Development, but at least with DOE and DOD, in working with the CCA Forum in defining reference standards for component-based computational frameworks. It should encourage and promote coordination between the CCA Forum and the Global Grid Forum.

  The success of software-component technology will depend on the creativity in both the design and implementation of systems and, thus, will depend on past and future government-supported research done at universities and government labs. However, no matter how good the technology, a significant and possibly primary benefit will be the creation of a synergetic software environment where codes can be easily shared. Such an environment must be based on standards, and this is where government agencies should play a significant role.

  While scientific software components offer significant benefits for programming efficiency and code sharing, a viable market is also needed to support the development cost. The small size of the scientific market cannot support high software development costs.

  An organized program to transition proven high-performance component technology and applications/user designs to industry is necessary to satisfy software life-cycle requirements (maintenance, support, and training) at NASA. Programming environments are large and generally evolving software systems. Development and maintenance costs are too large to be handled by a small or modest-sized laboratory. The open-source approach is good for including creativity in the early developmental stages of a software system, but will not provide the reliability needed to support large application projects. However, NASA as well as the scientific community, also cannot afford to support a single-vendor solution. New ideas leading to improved technology need to be implemented in a timely fashion to support research goals.

  Potential benefits to NASA in carrying out Recommendation 3 are as follows.

  - It will help NASA to identify its own requirements based on its applications and strategic vision.
  - It will provide incentive to commercial vendors to join the CCA Forum, which in turn will influence new framework products.
  - It can share the cost of the design of frameworks with other agencies that have similar applications and programming requirements.
  - It will enable NASA to reduce software development costs in the long run.
  - It provides a convenient vehicle for partnering with universities and industry in developing workable standards and best practices.

- Recommendation 4

  NASA should promote and support the adoption of scientific software component technology through an education program for application developers and users and a technology transition effort.

  As noted earlier, scientific programmers tend to develop their software independently, partly because programming tools for scientific computing environments have had little acceptance. If this is to change, the use of software-component technology will need to be not only encouraged, but supported by NASA and other agencies in the development of their mission-critical software. They will need to educate their application developers and users of the benefits of the new technologies. User education should include:

  - an understanding of the role of software-component technology in an overall environment where codes are shared,
  - training in the tools needed to create and use components,
  - an understanding of good code design for use with component-based frameworks and other tools,
  - a suggested transition strategy to change their programming styles, and
  - suggested strategies to include or to migrate old codes to the new environments.

  The Task Force suggested in Recommendation 2 above could play a vital role by collecting reasonably detailed user-based requirements that will help guide the design of new technologies and create a synergy between application developers and programming system developers leading to more adoption of new technology. It could organize workshops and hold seminars to promote a broader understanding of software-component technologies and their use. The task force could assist users and application developers in developing a transition strategy for converting old codes to the new technology beyond the use of simple wrapping tools and templates. To gain full benefits of the new technology, old codes should be split into appropriate modular pieces that best integrate into the technology. Most importantly, tools are needed for integrating new and legacy codes into new/bigger/more functional overarching multi-disciplinary applications.

5. Discussion Related to Recommendation 1.

- Good modular application design is important.

  Over the years scientific programmers have learned that good programming requires good organization of data and execution steps. In the early days, good programming was focused heavily on performance. More recently, code reuse and maintenance have become increasingly important. Good organization translates into a more understandable programming style. The trend toward distributed applications just increases the importances of good design. Distributed computing usually includes a wide range of communication performance. Appropriate organization and location of data to minimize data transfers can reap big performance gains. Also, the management of code becomes more difficult for distributed applications as the code elements must be grouped for appropriate and flexible distribution to the various computers being used.

  For applications designed to execute on a single computer, associated code elements can be managed with minimal co-ordination and the loader can link them together with good efficiency. For modern applications, related code elements will need to be built in a modular fashion so that computational frameworks can link the elements to form a composite application. The effectiveness of the framework will be limited by the quality of the modular design of the elements passed to it. One criteria is that such modules should be chosen to allow code elements that use the same data to be easily and efficiently grouped on one computer. However, distributed computing is used when large data sizes or performance needs dictate that approach. Modules also need to be designed to support efficient programming and execution of data transfers.

Components designed with good modularity support the previously mentioned advantages: rapid integration, plug-and-play capability, easier code validation and maintenace, and increased code reuse. A good modular design is the essence of a systematic format needed to build successful distributed applications.

- Programming flexibility enables good performance.

  Scientific applications can have a wide range of performance requirements, even within the same application. Once the appropriate modularity is chosen, a programming model is needed that allows the programmer to communicate the performance requirements of at least the critical modular elements (or components).

  The optimization of data flow is clearly one need. Historically, performance optimization has focused on organizing the location of data storage and orchestrating data transfers. The details of data storage and transfers are different for distributed and grid programming, but they are still important to performance. A programming model needs to provide flexible, easy-to-program abstractions that give the application developer sufficient control to create quality applications. For example, depending on the size of a data set and the frequency of its use, a programmer may choose to transfer data between different computers for use by different element applications. Alternatively, it might be more efficient to integrate several element applications into the same process for access to a data set that cannot be moved efficiently.

  Distributed applications will need a variety of performance solutions. Remote process creation, remote task execution, data transfers, event signals, and other remote operations will have different requirements for different applications. Even the choice of computer on which to execute a particular code can be important. And, the choices can be dynamic when code parameters such as data sizes are allowed to vary. A capability is needed which allows application developers to indicate performance requirements so that the underlying PSE can provide the appropriate implementation. One approach to achieving flexibility while supporting single-implementation components is via a filter strategy. For cases where the flexibility requirement relates to communication between different application elements, filtering software can be inserted between the relevant outputs and inputs.

  A component framework can result in decreased performance over a "hand-coded" solution that directly uses a low-level, high-performance communication system. However, prototypes are showing that the performance overhead can be kept small. The key is to define specifications and create implementations where appropriate information is passed from the application developer to the framework via metadata. This is also a key to support code reuse. Once code usage characteristics, such as performance hints, are packaged with the code to form a component, other users will be less reluctant to incorporate that code in their composite application.

- Metadata should not be limited to interface specifications.

  Interface specifications describe the calling arguments for the method or function being accessed from a component. This is the minimal information needed to use a component. Metadata can also include specifications related to internal code behavior. This allows a code developer to alert a potential user or composite application developer of important algorithm characteristics. When very large composite applications are built, the correct inclusion of all element applications into the composite application cannot be manually managed by traditional code inspection by the application developer. Sophisticated metadata schemes and contracts specifying composite application requirements will be needed to allow the framework to more accurately verify the many details of correctly building the composite application.

An aggressive use of metadata is needed for code reuse, plug-and-play capability, rapid integration, and other component advantages to reap full benefits. When 10's or 100's of element codes are merged to create a composite application, even a group of programmers will find it difficult to accurately analyze all aspects of integrating a large set of codes. Metadata/contract systems will be needed to reduce the amount of detail that the programmer directly analyzes. This is particularly true for distributed applications.

- Portability and interoperability support are still needed.

  The increased importance of code reuse and sharing translates to concerns about portability and interoperability. The component approach, particularly for distributed applications, can reduce one aspect of this problem. Instead of porting a code, one can just use remote process management and other distributed techniques to run the code on a friendly architecture. In some cases a code will not have to be ported across different architectures.

  However, other aspects of portability are still a concern. The benefits of using component methodology are increased when there are lots of compatible PSEs. Multiple component specifications and framework designs can create incompatibilities. Even if the single component specification is used, different vendor frameworks will implement different contorl and communication protocols if such protocols are not part of an interoperability standard. Thus, the use of component and distributed programming technology will shift portability concerns from architecture issues to framework issues. There is a significant amount of disagreement in the scientific community over what should be the nature of solutions to portability and interoperability. This may be a significant obstacle to the rapid development of software component technology for the scientific community.

  The number and nature of standards will significantly affect the benefits of component technology relating to code reuse, plug-and-play capability, and rapid integration.

- Standards should not stifle creativity.

  The wealth of knowledge, experience, and system infrastructure that results from all the recent Grid research can most effectively be leveraged if it were possible to pick features from each system and create a best-of-all-worlds system. Standardization is one way to achieve this level of portability and interoperability.

  On the other hand, the lifeblood of the scientific community is creativity. Scientific programming standards should, therefore, support continual research, development, and insertion of new programming technologies. This requires standards that are flexible enough to support continual experimentation while also providing programming efficiency. This is one reason for not directly adopting current business-based programming solutions.

  The benefits of component methodology will only be obtained if there is wide acceptance. Otherwise, they will be viewed as just another programming overhead that is best avoided. While the design process of open standards can be frustrating, the result will be better accepted by scientists and engineers. A logical consequence is that a single software or hardware vendor should not define scientific programming standards.

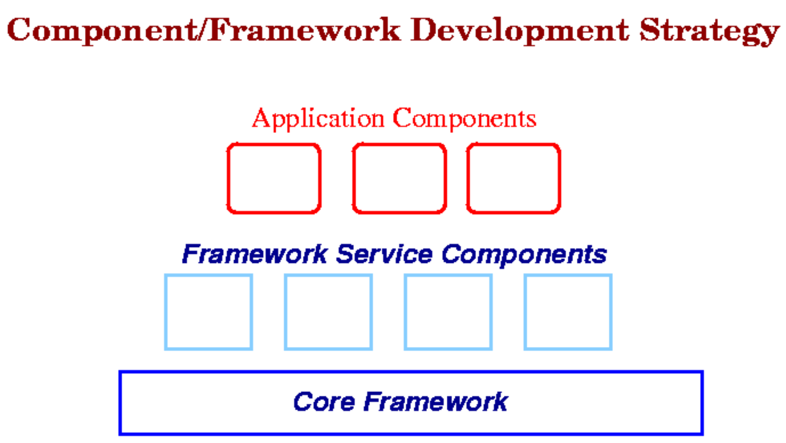
Component/Framework Development Strategy
Application Components
Framework Service Components
Core Framework

FIG. 6.1.

6. Discussion Related to Recommendation 2.

- Programming models are needed to bridge the gap between framework designs and application programming requirements.

  As mentioned in the Introduction, system software developers and application developers have not sufficiently exchanged ideas as to the exact nature of future programming systems. Lists of programming features and execution services have been shared, but this is not sufficient. Programming is an interactive task where the application developer communicates desired application concepts through the abstractions and rules of some programming language or system. These abstractions and rules, an implementation of some programming model, need to be carefully designed. If they are too complex, programmers will either not learn them or will become distracted from their primary task—implementing an algorithm that models some physics. If they are too simplistic, programmers will become frustrated with the inability to efficiently express the necessary application concepts. A sufficiently broad programming model needs to be defined that can be used as a guide for implementing software component technology.
- A development strategy for scientific components is needed that supports growth.

  The need for creativity and flexibility calls for a development strategy that can grow. Software component technology supports growth and the insertion of changes. However, a growth strategy should not only be applied to the development of scientific applications, but also to the development of a component/framework specification and to the evolution of frameworks from the prototype to the production stage.

  The development of scientific software components and frameworks should begin with as simple a design as possible, but one that captures the fundamental functionality. As shown in Figure 6.1, one would develop a core framework specification and implementation. Framework services would be added as components. This would allow various framework designs to be tested in parallel with the development of the initial application components. One of the major problems with getting good requirements for component-based frameworks is that most application developers lack the appropriate experience. A concurrent system and application software development strategy should help prevent application developers from being stuck with premature design decisions by system developers.

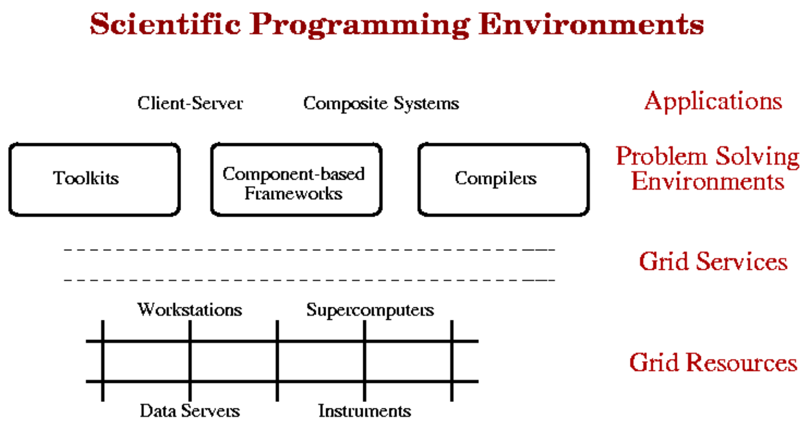

Fig. 6.2.

In addition, it is not clear how many styles of frameworks and service components are desired. A single-core computational framework implementation would appear ideal. But can it provide the range of functionality that scientists and engineers desire? Similarly, will one set of synergetic service components be sufficient?

This suggests that a set of core-design specifications be developed and supported by NASA. From this specification, a base reference implementation should be developed and supported. A set of core service components should also be implemented as part of the base reference implantation. After this the design process should be allowed to mature for a period of time. Based on feedback from early application and system software developers, the reference implementation can be modified and alternative implementations can be created.

- In the long run, the scientific component methodology should not focus on a single set of Grid services.

  The component-based framework recommended in this report is one type of Problem Solving Environment that can be used to develop and execute scientific applications in Grid environments. Figure 6.2 shows the relationship of the various software layers in such an environment. The software (OS, system libraries) used to access Grid resources will probably always have a heterogeneous nature. Various Grid service systems have been developed to unify these heterogeneous interfaces. Currently the Globus Toolkit [5] is being targeted by the [6] IPG project as the primary set of Grid services. Targeting one set of services is beneficial to the early development of software systems and even user applications. However, programming models and problem solving environment designs should not be tied to a specific set of services. A good implementation of a software system should either be relatively easy to port to a different set of Grid services or, even better, be configurable to support multiple sets.
- Multiple languages and architectures should be supported.

  A component/framework specification should support most languages and hardware architectures used by scientists and engineers. However, a single language is needed to define interface and behavioral specifications.
- The emphasis on graphical user interfaces should be secondary.

  Graphical programming environments are being increasingly used in the scientific community. Graphical interfaces are useful for all aspects of application development and usage including the programming of high-level work-flow. This is important for the development of applications by teams. High-level information can be disseminated more rapidly in graphical form. Additionally, such an environment can reduce the efforts needed to educate users about new programming models, and reduce the slope of the learning curve of new tools and the inevitable inertia to migrate to new and different paradigms.

  On the other hand, scientific-component technology is still in a development stage. Determining the best design for components to meet the requirements of the scientific community is a big challenge. Developing graphical interfaces is challenging in its own right as poor interfaces can deter usage. It is recommended that the emphasis be first placed on developing scientific component technology.

- Grid programming requirements such as trust and security should be supported.

  Trust and security also require programming flexibility. Security services can themselves be components, and scientific components (for performance reasons) must be allowed to rely transparently on the security services of their environment. That is, scientific programmers should not be burdened with understanding implementation details of network security.

- Proposed solutions and prototypes generally need to scale to large applications and to support different application designs.

  Many prototypes are used to give impressive demonstrations; especially when interesting graphics is included. Unfortunately, the applications used in the demonstrations are not representative of the large size or design of more complex applications. This is not just a result of deceit, but it is an artifact of limited development budgets. Also, complex applications that would best benefit from a sophisticated programming system have not been built because they are too costly to build without such a system—a "Catch-22". The bottom line is that evaluation of prototypes must be done carefully to consider the need for application growth.

- Batch as well as interactive programming environments need support.

  It is common for software systems to be developed in interactive environments. However, a significant amount of scientific applications are run in batch environments. As larger applications are developed, there will be a need to run using multiple batch systems along with interactive systems at the same time. Research on co-allocating resources is being conducted; one example being the Globus Project [5]. This is another example of a requirement that is often overlooked when evaluating prototypes.

- Business solutions should not be ignored.

  While the development and adoption of scientific standards is important, business-based solutions, such as CORBA, should not be ignored because such technologies are being used in developing some engineering tools. This calls for an approach that supports the integration of components based on the different technologies.

7. Summary: Future Plans. This workshop is just one step in the ICASE goal of researching solutions for modern Problem Solving Environments for NASA. The results of this workshop along with the requirements generated by application developer groups will be presented to NASA personnel. Feedback from NASA will be gathered during these presentations to refine the recommendations. The workshop results will also be shared with the Global Grid Forum, the CCA Forum, interested government laboratories and with other groups such as commercial companies.

Appendix: Related Projects and Products.

- Current Use of Components and Frameworks at NASA

Software component technology is being used in some NASA projects. This includes systems based on commercial technology developed for business applications and prototype frameworks based on component-like concepts. However, there is no organized, overall approach for applying component technologies to NASA scientific and engineering applications. Components provide the most benefits when they are used to support the sharing of code and data and this requires standards. Although some standards exist for business applications, a standards-based approach is needed for scientific applications. NASA should help in defining such standards and be a leader in applying modern software development practices to make sure that new codes are initially developed according to such practices rather than retrofitting them after the fact.

Component-like approaches were used in projects, such as [7] FIDO, at NASA Langley in the early 1990's. Here the emphasis was on learning the fundamentals of developing modular applications that could be used in distributed heterogeneous computing environments. The Multi-disciplinary Optimization Branch and the HPCC Offices have continually supported projects to build distributed, design-optimization applications. These application developments used several commercial packages (CORBA, Java RMI, iSIGHT, and Phoenix Model Center and Analysis Server). NASA also has supported third-party development via the SBIR Program (the LAWE prototype by High Technology Corporation [8]) and via support of research at ICASE (the Nautilus and Arcade projects).

- iSIGHT is a software framework that automates the tedious, repetitious job of running your design analysis programs. iSIGHT was developed by Engineous Software Inc. [9].
- Phoenix products and processes integrate and automate dataflow between critical applications across the enterprise. Model Center and Analysis Server were developed by Phoenix Integration [10].

The value of the component-based frameworks has also been recognized by NASA management as per the NASA ESS HPCC Round-3 request for proposals. It was noted, however, that most efforts emphasized the wrapping of legacy codes to turn them into components. NASA should strongly support modern software development practices to make sure that new codes are developed from the beginning according to such practices rather than retrofitting them after the fact, which reduces the benefits.

The Numerical Propulsion System Simulation (NPSS [11]) Project centered at NASA Glenn Research Center is developing an advanced engineering environment or integrated collection of software programs for the analysis and design of aircraft engines and, eventually, space transportation components. It accomplishes that by generating sophisticated computer simulations of an aerospace object or system, thus permitting an engineer to "test" various design options without having to conduct costly and time-consuming real-life tests. NPSS uses an object-oriented approach and incorporates a number of component concepts. Elements of an engine are modeled with codes packaged so that they fit together for a complete engine simulation. The component-like design allows for efficient interchange of element codes.

At NASA Ames (NAS Division), a component framework effort, Growler, supports environments that are distributed across heterogeneous platforms, including environments that require low-latency interactivity. Growler includes capabilities for collaborative visualization, analysis, and computational steering. To date, applications supporting several science domains of particular interest to NASA have been demonstrated. Another project is titled "Agent-based Integrated Analysis Framework." This is a prototype framework for integrating low- and high-fidelity analysis tools to conduct multidisciplinary analysis over distributed computing systems. The package includes interpolation and grid-generation tools to transfer data among CAD, CFD, and FEM analysis packages. A distributed computing capability was added using Analysis Server, a commercial software package. Prototype agent behavior was added to several components. An expert system was used to manage input data, select an appropriate flow, and build CFD flow solver input files.

- Commercial Technology Resources

  Commercial software-component technology (CORBA [12], Java Enterprise [13], Java Beans, [14], and DCOM [15]) has primarily been developed for business applications. This technology is discussed in [2]. These systems provide available tools, demonstrated success, and tested reference specifications, but these features fall short of meeting scientific requirements.

  Commercial technologies

  1. lack support for scientific data types,
  2. lack support for scientific programming languages,
  3. are not available for most supercomputer systems,
  4. generally exhibit poor runtime performance, and
  5. are too complex for easy adoption.

  Component technologies for scientific use should draw on the best features of commercial technologies, but be enhanced to meet the needs of scientific users.

  An example of a commercial effort that is targeting scientific applications is the FIPER Project that includes a consortium of companies along with limited government lab and university involvement. FIPER is documented in [1]. General Electric Corporate Research and Development is leading the project that has been funded by the National Institute for Standards (NIST-ATP). This effort is focused on developing a product around existing and emerging commercial technology. There is no guarantee that any resulting product will meet the requirements of scientific components highlighted below.

- Research Projects

  The CCA Forum effort has been mentioned as part of the above recommendations. At least four prototypes are associated with this effort.

  1. CCAFFEINE is a prototype of a design for Single Program Multiple Data (SPMD) computing with components based on the existing CCA specification. Development is based at Sandia National Laboratory [16].
  2. CCAT is a prototype, distributed, software-component system for scientific and engineering applications that is based on the CCA specification. Development is based at Indiana University [17].
  3. The Componentsllnl.gov project is focusing on developing high-performance language interoperability capabilities for scientific languages, including Fortran 77, Fortran 90, C, C++, Java, MATLAB, and Python. The Babel language interoperability tools and library are being developed in parallel with the CCA specification. [18].
  4. Uintah is a Problem Solving Environment targeted for applications that are part of the C-SAFE (Center for the Simulation of Accidental Fires & Explosions) project. Development is based at the University of Utah. [19].

Another example is the High Level Architecture (HLA [20]). The HLA is a general purpose architecture for simulation reuse and interoperability. The HLA was developed under the leadership of the Defense Modeling and Simulation Office (DMSO) to support reuse and interoperability across the large numbers of different types of simulations developed and maintained by the DoD. The HLA Baseline Definition was completed in 1996. The HLA was adopted as the Facility for Distributed Simulation Systems 1.0 by the Object Management Group (OMG) in November 1998. The HLA was approved as a standard through IEEE (IEEE Standard 1516) in September 2000. HLA is targeted for discrete event simulation in contrast to equation simulations in the scientific community. However, there are similarities.

The Advanced Programming Models (APM [21]) working group of the Grid Forum is working to identify existing efforts. A white paper, titled "Problem Solving Environment Comparisons," provides an overview of several prototype efforts.

Below is a list of related projects at various government laboratories and universities.

1. Cactus is a PSE designed with a modular structure to enable parallel computation across different architectures and to support collaborative code development. Development is based at the Albert Einstein Institute in Germany. [22]
2. DataCutter is a programming model and runtime support system intended for data-intensive applications that makes use of remote data sets. Development is based at the University of Maryland. [23]
3. Jaco3 is a software environment for coupling industrial simulation codes across the Grid using CORBA. Development is based at IRISA/INRIA in Paris. [24]
4. The Nautilus Project is developing a programming and execution environment for building large applications that execute on a computational Grid. Development is based at ICASE/NASA Langley Research Center. [25]
5. Nimrod/G is a tool for automated modeling and execution of parameter sweep applications over global computational Grids. Development is based at Monash University, Melbourne, Australia. [26]
6. Ninf is a network-enabled, RPC-based server system for numerical computing. Development is based at the Tokyo Institute of Technology. [27]
7. OpusJava is a Java-based framework for distributed, high-performance computing that provides a high-level component infrastructure and that facilitates the seamless integration of HPF modules into a larger distributed environment. Development is based at the Institute for Software Science, University of Vienna. [28]
8. Netsolve is a project that aims to bring together disparate computational resources connected by computer networks. It is a RPC-based client/agent/server system that allows one to access both hardware and software components. [29]
9. Arcade is a web-based environment to design, execute, monitor, and control distributed applications. [30].

A related system is Legion, being developed at the University of Virginia. Legion is an integrated architecture that provides a programming environment for the development and execution of applications in Grid environments. Legion includes lower-level, distributed computing services that are not part of most of the PSE's mentioned in the report [31].

REFERENCES

[1] P. Rohl et al., A Federated Intelligent Product Environment, AIAA-2000-4902, 2000.

[2] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[3] Global Grid Forum, www.gridforum.org.

[4] Common Component Architecture Forum, www.cca-forum.org.

[5] Globus Toolkit, www.globus.org.

[6] IPG project, www.ipg.nasa.gov.

[7] FIDO, hpccp-www.larc.nasa.gov/∼fido/homepage.html.

[8] LAWE, www.htc-tech.com/parallel.

[9] Engineous Software Inc., www.engineous.com.

[10] Phoenix Integration, www.phoenix-int.com.

[11] The Numerical Propulsion System Simulation, hpcc.larc.nasa.gov/npssintro.shtml.

[12] CORBA, www.omg.org.

[13] Java Engerprise, java.sun.com/products/ejg.

[14] Java Beans, java.sun.com/products/javabeans.

[15] DCOM, www.microsoft.com/com/tech/dcom.asp.

[16] CCAFFEINE webpage, Sandia National Laboratory, www.cca-forum.org/ccafe.html.

[17] CCAT webpage, Indiana University, www.extreme.indiana.edu/ccat.

[18] CCA, www.llnl.gov/CASC/components.

[19] Uintah, Center for the Simulation of Accidental Fires & Explosions (C-SAFE), www.csafe.utah.edu.

[20] High Level Architecture, www.dmso.mil/index.php?page=64/hla.

[21] Advanced Programming Models, www.eece.unm.edu/∼apm.

[22] Cactus, Albert Einstein Institute, Germany, www.cactuscode.org.

[23] DataCutter, University of Maryland, www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm.

[24] Jaco3, IRISA/INRIA, Paris, www.arttic.com/projects/JACO3.

[25] The Nautilus Project, ICASE/NASA Langley Research Center, www.icase.edu/∼teidson/Nautilus.

[26] Nimrod/G, Monash University, Melbourne, Australia, www.csse.monash.edu.au/∼davida/nimrod.html.

[27] Ninf, Tokyo Institute of Technology, ninf.etl.go.jp.

[28] OpusJava, Institute for Software Science, University of Vienna, www.par.univie.ac.at/∼erwin/opus.

[29] Netsolve, www.cs.utk.edu/netsolve.

[30] Arcade, www.icase.edu/arcade.

[31] Legion, legion.virginia.edu.